A Multi-tenant Key-value SSD with Secondary Index for Search Query Processing and Analysis

DONGHYUN MIN, KIHYUN KIM, and CHAEWON MOON, Dept. of Computer Science and Engineering, Sogang University, Republic of Korea
AWAIS KHAN, Oak Ridge National Laboratory, USA
SEUNGJIN LEE, Dept. of Computer Science and Engineering, Sogang University, Republic of Korea
CHANGHWAN YUN and WOOSUK CHUNG, Memory System R&D, SK Hynix
YOUNGJAE KIM, Dept. of Computer Science and Engineering, Sogang University, Republic of Korea

Key-value SSDs (KVSSDs) introduced so far are limited in their use as an alternative to the key-value store running on the host due to the following technical limitations. First, they were designed only for a single tenant, limiting the use of multiple tenants. Second, they mainly focused on designing indexes for primary key-based searches, without supporting various queries using a combination of primary key and non-primary attribute-based searches. This article proposes Cerberus, a Log Structured Merged (LSM) tree-based KVSSD armed with (1) namespace and performance isolation for multiple tenants in a multi-tenant environment and (2) capability for processing non-primary attribute-based search queries. Specifically, Cerberus identifies the tenant's namespace and splits a single large LSM-tree into namespace-specific LSM-tree indexes for tenants. Cerberus also manages secondary LSM-tree indexes to enable non-primary attribute-based data access and fast search query processing. With the SSD-internal CPU/DRAM resources, Cerberus supports non-primary attribute-based search queries and handles complex queries that are combined with search and computing operations. We prototyped Cerberus on the Cosmos+ OpenSSD platform. When there are multiple tenants, Cerberus exhibits up to 2.9× higher read throughput and negligible write overhead compared to existing KVSSD. Cerberus also shows lower latency by up to 9.31× for non-primary attribute-based queries.

CCS Concepts: • **Information systems** → *Database management system engines*; *DBMS engine architectures;*

Additional Key Words and Phrases: Key-value solid-state drive, NoSQL database storage engine

This work was funded in part by SK hynix research grant, and in part by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. NRF-2021R1A2C2014386).
This is a revised version. A preliminary version of this work was published in the 2021 ACM Hot Topics in Storage and File Systems (HotStorage'21).
Authors' addresses: D. Min, K. Kim, C. Moon, S. Lee, and Y. Kim, Dept. of Computer Science and Engineering, Sogang University, Seoul, Republic of Korea; emails: {mdh38112, realltd16, mcy98, seungjinn, youkim}@sogang.ac.kr; A. Khan, Oak Ridge National Laboratory, Oak Ridge, TN; email: khana@ornl.gov; C. Yun and W. Chung, Memory System R&D, SK Hynix; emails: {changhwan.youn, woosuk.chung}@sk.com.

1539-9087/2023/07-ART65 $15.00
https://doi.org/10.1145/3590153

ACM Reference format:
Donghyun Min, Kihyun Kim, Chaewon Moon, Awais Khan, Seungjin Lee, Changhwan Yun, Woosuk Chung, and Youngjae Kim. 2023. A Multi-tenant Key-value SSD with Secondary Index for Search Query Processing and Analysis. *ACM Trans. Embedd. Comput. Syst.* 22, 4, Article 65 (July 2023), 27 pages.
https://doi.org/10.1145/3590153

1 INTRODUCTION

Key-value Store (KV-Store) is a database that manages key-value pairs running on a host. The KV-Store has been widely employed in many modern applications because of its simple key-value interface [4, 13, 15, 25, 29, 42]. On the other hand, **Key-value Solid-state Drive (KVSSD)** is an SSD with a key-value interface that runs the storage engine of the KV-Store on the SSD [7, 12, 20, 22, 31, 46]. Unlike KV-Store, KVSSD not only saves host-side CPU and DRAM resources but also reduces I/O amplification. Among various KVSSDs, Pink [20] and iLSM-SSD [31] are the latest KVSSDs that adopt a **Log-structured Merge (LSM)** tree-based indexing approach. LSM-tree [36] is a data structure that performs out-of-place logging to disk sequentially, instead of in-place write when storing KV pairs. Therefore, the LSM-tree is used as the primary index of the KVSSD's storage engine to maximize write performance.

Most of the LSM-tree-based KVSSDs introduced so far assume a design for single tenant/user.[1] In other words, they do not support multi-tenancy, which is an architecture that can host database instances of multiple tenants on a server. In a multi-tenant environment, each tenant is provided with an abstraction of having their own dedicated database servers, which require isolation in terms of security, privacy, and performance [17]. The isolation requirement can be met based on namespace isolation. Because namespace isolation separates all tenants' KV pairs stored in a physical space into multiple logical groups, each tenant is provided an isolated view of their own database instance. Traditional host-side KV-Stores such as RocksDB [15] and MongoDB [4] have been designed to be multi-tenant, satisfying the aforementioned isolation requirements [30, 41]. On the other hand, current LSM-tree-based KVSSDs lack a design to support namespace isolation. Therefore, the KVSSD fails to provide tenants with the strict view, showing only data corresponding to their own namespaces. If the KV pairs using the same key exist between tenants in KVSSDs, the KV pairs of other tenants can be recklessly modified or read if they are not carefully managed [21, 38].

Furthermore, LSM-tree-based KVSSDs in a multi-tenant environment do not provide as much promised read throughput as embedded devices can provide for each tenant. This is because multiple KV pairs from tenants are still managed by a single global-shared LSM-tree index structure. The single global-shared LSM-tree structure of the KVSSD has the following limitations [35]. First, index entries of certain tenants' KV pairs are frequently pushed to the higher level in the LSM-tree due to the index entries of other tenants' KV pairs. From each tenant perspective, traversing the LSM-tree is time consuming because the search algorithm of the LSM-tree begins at the lowest level of the tree. Second, to check the existence of the requested key during the LSM-tree search, it must read the LSM-tree's **Bloom Filter (BF)** data structures stored in the NAMD flash memory, resulting in multiple NAND flash read operations.

Last but not least, the value part in a KV pair is rich in attribute information and **primary key (*pk*)**-based search alone cannot fully meet the desire to mine such value rich information. Thus, KVSSDs only maintaining a *pk*-based index are limited in providing such value-based or non-primary attribute-based search and query services. In contrast, several LSM-tree-based KV-Stores such as Cassandra [29], MongoDB [4], and DynamoDB [42] support searches based on

[1]Originally, the tenant is a group of users who share common access to the database instance. In this article, the tenant and user terms are used interchangeably for simplicity.

non-primary attributes as well as *pk*. They allow users to access any attribute field in the database to meet various search and query processing requirements. To this end, several KV-Stores additionally set the specific non-primary attribute to a **secondary key (*sk*)** and provide a separate secondary LSM-tree index centered around the *sk*. The *sk* is chosen among any non-primary attribute fields stored inside the value part of an entry, which may be in the JSON format [25, 50]: *Value* $v = \{A_1 : value(A_1), \ldots, A_n : value(A_n)\}$, where the $value(A_i)$ is a value of the non-primary attribute A_i. Each entry in the secondary index is a pair of *sk* and *pk*, which acts as a role of key and value, respectively. In the KV-Store, users can request queries with various search predicates, which are a user-provided condition on the *sk* basis. The KV-Store can find and return the *pk* list of the KV pairs that satisfy the search predicate by using the secondary index with *sk*. Then, the user can read actual KV pairs of the *pk* list by using a primary index. Accordingly, checking all KV pairs is not required in the KV-Store. On the other hand, LSM-tree-based KVSSDs only manage a *pk*-based primary LSM-tree index. Thus, if a user wants to retrieve KV pairs satisfying a specific *sk*-based search predicate, the user must retrieve all KV pairs by using the primary index, load them all into the host's memory, and weed unsatisfied KV pairs out by checking the search predicate. This approach imposes a significant data transmission cost for all KV pairs from the KVSSD to the host. The approach also blocks an opportunity to utilize the in-storage computation capability of KVSSD. Therefore, in order for the KVSSD to be used as an alternative device for the storage engine, the support for *sk*-based search is required.

In this article, we propose Cerberus, an LSM-tree-based multi-tenant KVSSD that (1) provides namespace and read performance isolation by providing a per-namespace LSM-tree index design and (2) supports *sk*-based search query processing capability by providing secondary LSM-tree indexes. The specific contributions of the article can be summarized as follows:

(1) For namespace isolation, Cerberus checks the tenant's namespace information and performs primitive namespace-based access control for each tenant. Each tenant is assigned a unique namespace ID. The namespace ID is stored in the NSID region of the NVMe command and then transferred to Cerberus. With the namespace isolation, each tenant can be offered an isolated view on their KV pairs, which can be a primitive of security and privacy.
(2) For performance isolation, Cerberus adopts a namespace-dedicated LSM-tree design. Specifically, Cerberus provides each tenant with an independent and dedicated index by partitioning the single global-shared LSM-tree into per-namespace LSM-trees for each tenant. The per-namespace LSM-tree structure lowers the search depth of the LSM-tree. Thus, Cerberus delivers as promised read performance to tenants as the KVSSD can provide.
(3) Cerberus manages secondary indexes for handling *sk*-based query requests. Cerberus allows a user to search for a *pk* list of the KV pairs, satisfying the *sk*-based search predicate by consulting the secondary index. Then, the user is able to read only necessary KV pairs by using the primary index. In order to further streamline the search process of the *sk*-based query, we propose a **Value Addressable Secondary (VAS)** index, which allows to directly read actual KV pairs and return them to the user by consulting only the secondary index. Therefore, Cerberus can completely eliminate the step of searching the primary index while processing the *sk*-based search query.
(4) The query with the *sk*-based search predicate can be vastly fused with the computing operation. For example, a search query can be fused with analytic operations (e.g., sort). To cope with these fused analytic queries, Cerberus utilizes both the VAS index and KVSSD's internal resources to perform both search operations and **In-Storage Processing (ISP)** operations.
(5) Cerberus is implemented on the Cosmos+ OpenSSD platform [2] in a Linux environment. We evaluate and compare Cerberus with the state-of-the-art LSM-tree-based KVSSD (iLSM [31]). Our extensive evaluations confirm the following: (1) Cerberus improves read throughput by

up to 2.9× while write performance is barely different from the baseline in the multi-tenant environment, (2) Cerberus improves the speed of *sk*-based search query processing by up to 9.31× compared with the baseline, and (3) for the *sk*-based search query fused with a compute operation (e.g., sort), Cerberus showed 3.9× reduced data transmission time compared to the case when each query is processed individually on the host.

2 BACKGROUND

2.1 Log-structured Merge-tree

Log-structured Merge-tree (LSM-tree) [36] is a hierarchical structure that consists of a memory component (MemTable) and a disk component (SSTable). MemTable temporarily stores the KV pair transferred from the user. MemTable is mutable and is typically implemented as a skiplist. An SSTable is an immutable index file created when KV pairs are flushed and stored from MemTable to the disk. This process is called SSTable flushing.

SSTable is structured into multiple levels ($L_0, L_1, \ldots, L_n$). Each level contains several key-sorted SSTables. In the LSM-tree, following the key-value separation design of WiscKey [33], each SSTable is composed of a metadata (key, value log offset) region and BF region. The metadata region consists of the key and the offset of an associated value stored in the value log. The value log acts as a write-ahead log structure, where KV pairs are physically appended. The BF is a memory-efficient probabilistic data structure and is used to test whether an element is a member of a set. It is known that large-scale databases can use 10% or more of their RAM memory for storing the BF [11, 15]. If the portion of BF is cached on RAM, then the existence of the target key may be verified without loading and checking all the keys of the SSTables from the storage. The L_i SSTable acts as a buffer for L_{i+1}, which is larger in size compared to the L_i. LSM-tree triggers the compaction process when the size of L_i reaches a certain threshold. Assume that the size of L_i has reached a certain threshold. Then, one L_i SSTable from L_i is selected as the victim, and the L_{i+1} SSTables with overlapping key ranges of the L_i SSTable are selected. The SSTables selected as victims are merge-sorted, and a new L_{i+1} SSTable is created and written to disk. Afterward, the victim SSTables that participated in the merge-sort are invalidated and they become free space for each level.

In the cases when key-value separation is applied to the LSM-tree, the entire values of KV pairs in the SSTable are not unnecessarily read and written, while I/O or internal operations (e.g., compaction) are performed on the SSTable. Thus, it reduces the I/O amplification compared to the original LSM-tree.

2.2 LSM-tree-based KVSSD

KVSSDs exploit internal resources (CPU and DRAM) to implement an LSM-tree index inside the SSD [7, 12, 20, 22, 31, 46]. Among them, iLSM-SSD [31] was the first LSM-tree-based KVSSD to implement the key-value separation technique. Figure 1 shows the overall system architecture of iLSM-SSD. The iLSM-SSD consists of value log, MemTable, and SSTables. In the MemTable and SSTables, keys and value log offsets are recorded and managed. The value log offsets are allocated and managed at the granularity of logical block addresses, through which the KVSSD can access the physical location of a KV pair in the NAND flash memory.

The operation process of iLSM-SSD is as follows. First, when the iLSM-SSD receives a Put() command, the iLSM-SSD stores a KV pair in the value log (❶). Then, the Put() command is completed after the key and the value log offset of the KV pair are inserted into a MemTable (❷). When the size of a MemTable exceeds the pre-defined threshold, the iLSM-SSD transforms the MemTable into a form of an SSTable and flushes it to L_0 (❸). As Put() commands are processed, the iLSM-SSD performs compaction in the same manner as a traditional LSM-tree. As a result, the tree structure of SSTables is formed inside the NAND flash memory.

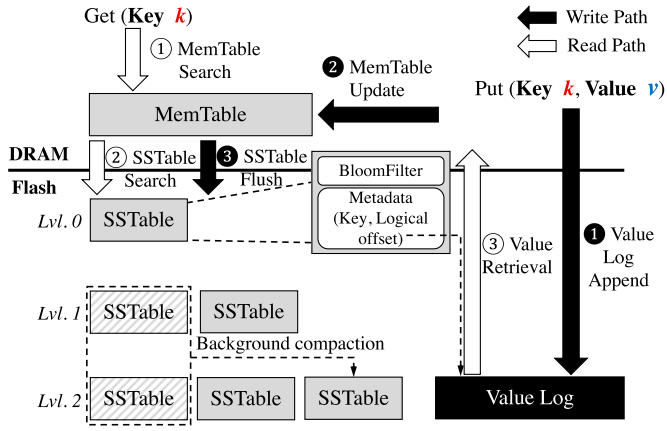


Fig. 1. An architecture of the state-of-the-art LSM-tree based KVSSD [31] that implements a key-value separation technique [33] inside an SSD. The figure depicts both KV write path (Put() command) and KV read path (Get() command).

Next, once the iLSM-SSD receives a Get() command with a target key, iLSM-SSD searches the MemTable first to check whether the requested target key exists (①). If the key does not exist in the MemTable, the iLSM-SSD begins searching SSTables to find the key from the lowest level (e.g., L_0) to higher levels (②). During the search, the iLSM-SSD first loads a BF that exists in each SSTable from the NAND flash memory to DRAM. After loading the BF, the iLSM-SSD performs a membership test with the BF to check if the requested key exists in the SSTable. If a membership test is successfully passed, the iLSM-SSD loads the remaining metadata region from the NAND flash memory to DRAM for actual key comparison. If the key of the metadata region matches with the requested key, the iLSM-SSD reads the value from the NAND flash memory by referring to the value log offset stored in the metadata region (③).

2.3 Motivation

2.3.1 Lack of Namespace and Performance Isolation. There have been two general methods for modeling index-level multi-tenancy [3, 5, 52]. The first method is an index-per-tenant model. In this model, multiple tenants occupy queries where each tenant has their own index. Accordingly, the database achieves data isolation. The second method is to use the multiple-tenants-per-index model. In this model, multiple tenants can share a single shared index and use tenant identifiers (e.g., namespace) to achieve an isolation of data access level. It is easy to manage the index because increasing the number of tenants itself does not require additional indexes. On the other hand, the recent LSM-tree-based KVSSDs lack design for these methodologies for data isolation and do not even support namespace-based data access isolation [20, 31, 46]. This causes the problem of KV pairs being incorrectly read or modified if multiple users have the same key [21, 38]. Also, the design of such an LSM-tree-based KVSSD that is shared by multiple tenants is not sufficient in terms of read performance isolation. For example, in a traditional LSM-tree-based KVSSD, multiple tenants share the single large LSM-tree index. Thus, the read performance of each tenant cannot be guaranteed.

In order to quantitatively analyze the degradation of read performance of each tenant, we conduct experiments for the following scenarios. The detailed experimental setup is described in Section 5.

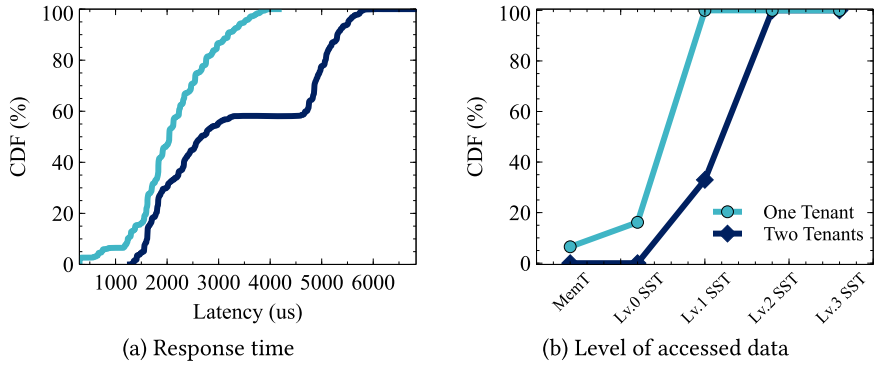


Fig. 2. Comparative evaluation of the impact that tenant x receives from co-located neighbor tenant y.

- Scenario 1: When the only tenant x's 1 million data monopolizes an LSM-tree.
- Scenario 2: When an LSM-tree is shared by tenant x's and tenant y's KV pairs. Each tenant has 1 million KV pairs and they access them at the same time.

For these two scenarios, Figure 2 shows the read performance and the statistic for the level at which the KV pair was found in the LSM-tree in terms of tenant x's perspective. We controlled the other experimental factors equally between two scenarios such as the size and the number of KV pairs. Specifically, Figure 2(a) compares the time required for tenant x to read 1 million KV pairs from KVSSD in these two scenarios. In Figure 2(a), the total average latency was increased by 1.58× in Scenario 2 compared to Scenario 1. This is because all tenants' KV pairs are indexed in a global-shared LSM-tree. In particular, there are two causes of performance interference between tenants. First, many KV pairs of each tenant are indexed at the higher level of the LSM-tree. Figure 2(b) is a comparison of the level distribution in which tenant x's 1 million KV pairs are indexed in the Scenarios 1 and 2. In Figure 2(b) and the case of Scenario 2, it is shown that the index entries of the KV pairs of tenant x were pushed to the higher level in the LSM-tree due to the KV pairs of tenant y. Compared to Scenario 1, about 67% of index entries of the KV pairs of tenant x were pushed down to SSTables at L_2 in the case of Scenario 2. Second, the number of loading BFs from the NAND flash memory into DRAM has increased during the search process. It was confirmed that the number of BF loads has increased by about 77% in the case of Scenario 2 compared to Scenario 1.

2.3.2 Lack of Search Query Using Secondary Index. Most of the current NoSQL systems provide a stand-alone secondary index [34, 37]. Figure 3 shows the schematic view of both the primary index and stand-alone secondary index. In the secondary index, the row key is a non-primary attribute's value, and the value is a list of row keys of the primary index. If the user requests a search query to a secondary index with a non-primary attribute's value as an *sk*, then the user receives the keys of the primary index. Then, the user looks up each of the primary indexes by keys and reads the actual values. For instance, the primary index for *User* and the secondary index for a *rating* column attribute are maintained in Figure 3. Notice the secondary index entry for *5* points to two index entries of the primary index. However, the current LSM-tree-based KVSSDs [20, 31, 46] cannot process search query with secondary key-based search predicates due to the following two reasons. First, current KVSSDs do not manage the *sk*-based secondary index. Thus, the KVSSD necessarily reads the entire KV pairs using the *pk*-based secondary index. Second, KVSSDs do not support the computational logic for data filtering. They have no logic to extract *sk* from the entire value and to check whether the *sk* satisfies the user-provided predicate. Thus, if the user wants to

read the KV pairs that satisfy a specific *sk*-based search predicate, the user inevitably depends on the primary index and manual predicate check. Specifically, the user performs the following steps. First, the user has to read all KV pairs from the KVSSD to the host by using the primary LSM-tree index. Then, the user manually extracts the attribute field from the value. Finally, the user checks whether the value of the attribute corresponding to *sk* matches to a given predicate filter.

To quantitatively analyze the performance overhead of the *sk*-based search query processing on the current LSM-tree-based KVSSD, where the only *pk*-based primary index exists, we compared the following two approaches:

- Approach 1: When the KVSSD does not support the secondary index. In this case, users inevitably read entire KV pairs to the host and then check the non-primary attributes from them manually.
- Approach 2: When the KVSSD supports the *sk*-based secondary index. In this case, the non-primary attribute field can be checked directly without checking all KV pairs inside an SSD.[2]

We implement KVSSD with the above two approaches in Cosmos+ OpenSSD [2]. The detailed experimental setup is described in Section 5. In order to evaluate these two approaches, we used the Yelp review dataset [49], which has around 200,000 KV pairs with the average value size of 0.78KB. The semantic of the used search query on the Yelp review dataset is the following:

> EXAMPLE OF SEARCH QUERY SEMANTIC. *Find the Comment (= attribute A_1) of the User ID (= pk) satisfying the predicate [Review Rating (= attribute A_2 used as a sk) is equal to 5 pt].*

In Approach 1, the KVSSD only uses a *pk*-based primary index to access all KV pairs in the dataset and transfers them to the host. The access to each KV pair requires a primary LSM-tree index search. During the primary LSM-tree index search, the KVSSD reads the BF and the meta-data of the SSTable from the NAND flash memory and performs a BF membership test and key comparison (Step 1). Then, the KVSSD reads actual KV pairs from the NAND flash memory (Step 2) and transfers them to the host (Step 3). Finally, the host checks if the *Review Rating* attribute of each KV pair is five or not (Step 4). Then, the host can extract *Comments* against the requested query.

In Approach 2, on the other hand, it is possible to search for an *sk*-based secondary LSM-tree index (Step 0). The KVSSD finds the *pk* list with the *Review Rating* value of five by consulting the *sk*-based secondary index. Next, in order to find the NAND flash memory address of the KV pair, the KVSSD performs the *pk*-based primary index search only on the found *pk* list (Step 1). Then, the KVSSD reads actual KV pairs from the NAND flash memory (Step 2) After the KV pairs are transferred to the host (Step 3), the host extracts *Comment* (Step 4).

Figure 4 shows the time breakdown analysis of the total execution time for processing the aforementioned search query with the *sk*-based search predicate for those two approaches. The total execution time for Approach 1 is 2.16× longer than that for Approach 2. In Approach 1, since the KVSSD does not have an *sk*-based secondary index, it is inevitable to read all KV pairs from the KVSSD to the host. The major overhead is a time to search for the *pk*-based primary index to read all KV pairs, which accounts for 88.8% of the total execution time. On the other hand, for Approach 2, the *pk*-based primary index search is performed only on the found *pk* list by using

[2]Most of the popular LSM-tree-based NoSQL systems (e.g., MongoDB [4] and Cassandra [1]) provide stand-alone secondary indexes. Thus, we also implemented a stand-alone secondary indexing technique [37] on KVSSD. The configuration setting of the secondary index is equivalent to primary LSM-tree index.

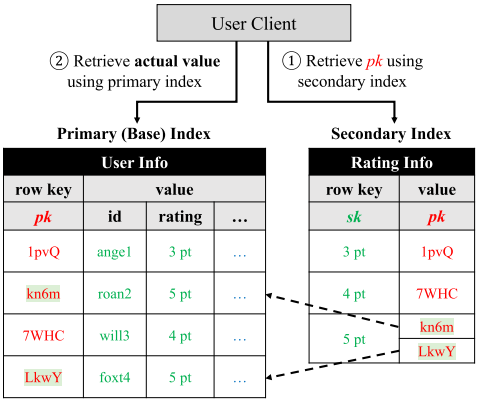


Fig. 3. A table-based schematic view of a secondary index and querying using a secondary index.

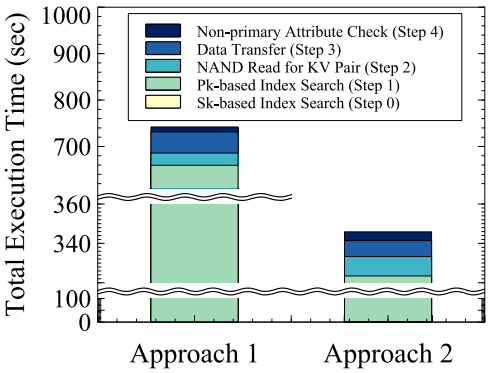


Fig. 4. A time breakdown analysis for Approach 1 and Approach 2. The size of KV pairs transferred and transfer time between host and KVSSD are the following: Approach 1 (1.5 GB, 45 sec), Approach 2 (2.5 MB, 8 sec).

the secondary index. As a result, the *pk*-based index search time is drastically reduced by up to 50.9% compared to Approach 1. This reduced overhead is proportional to the reduced number of the *pk*-based primary index search. In Approach 2, the total amount of KV pairs transferred to the host decreases by up to 2.3× compared with Approach 1. Also, the time for extracting *Comment* from the entire value decreases by up to 5.15× compared with Approach 1. In Approach 1, *Review Rating* and *Comment* should be extracted from all KV pairs. On the other hand, in Approach 2, the *sk*-based secondary index keeps the *pk* list of the KV pair whose *Review Rating* attribute is exactly 5. Thus, only *Comment* is extracted from the KV pair corresponding to the found *pk* list.

Based on these research motivations, this article proposes Cerberus, which aims to (1) isolate tenants' KV indexes according to the namespace, (2) optimize read performance for the multi-tenant KVSSD, and (3) provide service for search query with an *sk*-based search predicate by using a secondary indexing technique.

3 CERBERUS SYSTEM

3.1 Overview

Cerberus is a KVSSD that supports namespace and performance isolation between tenants in a multi-tenant environment and supports *sk*-based search. Figure 5 shows the overall architecture of Cerberus. For namespace isolation in Cerberus, tenants are given a unique namespace ID when creating a DB instance. After that, the users of tenants only access their own KV pairs by using the allocated namespace ID of the corresponding DB instance. NSTable in Figure 5 represents the schematic view of namespace-related information. In addition, in Cerberus, a single global-shared LSM-tree is partitioned and managed individually for each tenant's DB instance. This allows Cerberus to support read performance isolation and namespace isolation between tenants. At the same time, Cerberus manages the LSM-tree-based **Value Addressable Secondary (VAS)** index along with the primary index to support *sk*-based search. The users can request *sk*-based searches to the KVSSD, where the corresponding KV pairs can be searched through the VAS index. The key design elements of the Cerberus are outlined below.

- ***Per-namespace Dedicated LSM-tree Index:*** Each tenant is assigned a **Namespace ID (NID)** when creating its own DB instance. Next, the tenant makes a query request to the KVSSD together with the NID. The KVSSD internally maintains the per-namespace

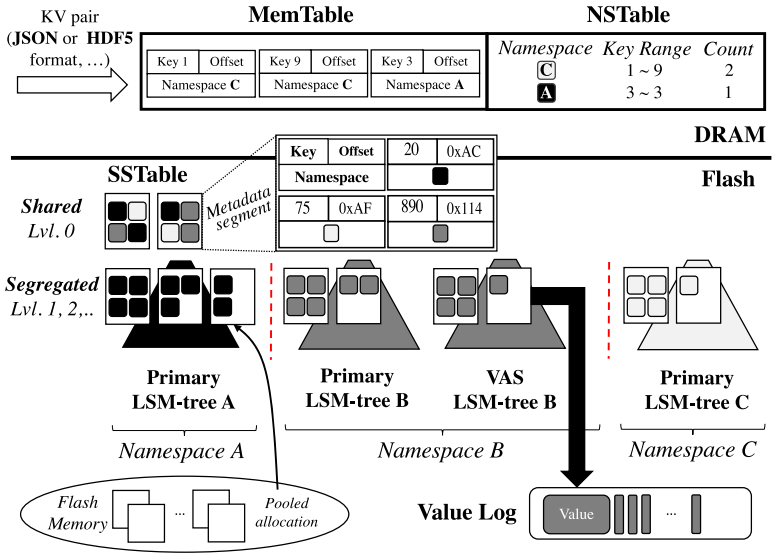


Fig. 5. An architectural overview for Cerberus supporting namespace and performance isolation for multi-tenant KVSSDs using Per-namespace LSM-tree index and *sk*-based search queries using secondary indexes. The VAS denotes Value Addressable Secondary index.

dedicated LSM-tree index separated according to namespace. Through this, each user/tenant manage KV pairs separately from other users. In addition, since the per-namespace LSM-tree has a relatively lower depth than the existing single global-shared LSM-tree, it is expected to support higher and guaranteed read performance.

- ***Value Addressable Secondary (VAS) Index:*** The VAS index uses the specific non-primary attribute value as the *sk*. The non-primary attribute means any column attribute not used as *pk*. An *sk* means a specific non-primary attribute value to be used in requesting and processing queries in column attributes. In the VAS index, the actual value of the *sk* is a pair of the *pk* and value log offset of the original KV pair. Through the VAS index, the KVSSD can process *sk*-based search queries. In addition, unlike the existing secondary indexing technique, the VAS index can directly access the KV pair by referring to the stored value log offset. Accordingly, the VAS indexing technique accelerates the processing of search queries that need to return KV pairs to users.

Together, these constructs enable multiple users of tenants to use the KVSSD as their own independent DB instances and become a building block to process search queries with *sk*-based search predicates.

4 DESIGN AND IMPLEMENTATION

This section first formulates the problem definition for two limitations of the existing LSM-tree-based KVSSDs discussed in Section 2.3. It then introduces a per-namespace dedicated LSM-tree index design for namespace and performance isolation. Lastly, it introduces a VAS index design for processing non-primary attribute-based search queries.

4.1 Problem Formulation

As mentioned in Section 2.3, the first limitation of current KVSSDs is that, when multiple tenants' KV pairs are indexed in the global shared LSM-tree, the time to access the KV pair of each tenant

Table 1. Notations Used and Their Descriptions

Notion	Description
n	The configured total level of LSM-tree ($0 \leq n$).
i	An index level of LSM-tree ($0 \leq i \leq n$).
HR_i	Percentage of KV reads that were processed at the level i SSTable.
HT_i	Time that was taken at level i for reading searched KV. This is a total latency combined of reading BF, metadata, and value of level i SSTable from NAND flash memory if key matches.
MP_i	Additional time required at level i due to miss. This is a total latency combined of reading BF and metadata of level i SSTables from NAND flash memory.
$MemTbl$	Time that was taken for accessing MemTable.
p	The total number of primary keys associated with requested secondary key ($p \geq 0$).

increases compared to when accessing it alone. There are two causes of this limitation. The leading cause is that many KV pairs stored by each tenant may be pushed down by KV pairs of other tenants and indexed at a deeper level in the LSM-tree. Another cause is that it requires loading BF from the NAND flash memory into DRAM multiple times during the LSM-tree search process.

To understand the problem of the LSM-tree, we build the cost model for formalizing each of the limitations mentioned above. Table 1 shows a summary of the notations and their descriptions in the cost model. The average latency for a KV pair access (*ALKA*) in the KVSSD can be estimated as a cost model in Equation (1) for accessing each KV pair indexed in the primary LSM-tree or the secondary LSM-tree index:

$$\begin{aligned} ALKA_i &= HR_i \cdot HT_i + (1 - HR_i)(MP_i + ALKA_{i+1}) \\ ALKA &= ALKA_0 + MemTbl. \end{aligned} \tag{1}$$

In Equation (1), $ALKA_n = HR_n \cdot HT_n$ in the last tree level n, assuming that Get() is not requested on the KVSSD where the requested key does not exist. This recurrence relation is modeled based on an insight of the average memory access time model [18] in the multi-level memory hierarchical architecture. This is because the LSM-tree is also a hierarchical structure of multi-level SSTables, and the LSM-tree search algorithm begins from the lowest level in order.

From Equation (1), the time delay for accessing the KV pair occurs when multi-tenants share a single large shared LSM-tree index that corresponds to the recursion execution frequency of MP_i. The second cause corresponds to the MP_i value itself in Equation (1).

The second limitation is that there is no design for secondary index and no logic for processing *sk*-based search queries in current KVSSDs. Therefore, in order to process *sk*-based search queries, all KV pairs must be loaded into the host memory. Even if the KVSSD adopts the secondary indexing technique of KV-Stores, the traditional secondary indexing method also has a limitation in reducing the KV pair access time. This is because the search for the primary index is also required to find the location of each value corresponding to each *pk* in the *pk* list identified through the secondary index.

The access time to all KV pairs required for processing an *sk*-based search query is the sum of each *ALKA* and can be expressed as shown in Equation (2):

$$\sum ALKA = (ALKA^{primary} \cdot p) + ALKA^{secondary}. \tag{2}$$

In Equation (2), $ALKA^{primary}$ and $ALKA^{secondary}$ mean KV pair access times through primary and secondary index, respectively. Particularly, when processing an *sk*-based search query, p in Equation (2) increases proportionally, as the number of all associated *pks* found from the requested *sk* increases.

4.2 Per-namespace LSM-tree Index

The per-namespace dedicated LSM-tree index logically separates each tenant's KV pairs and allows the KV pairs to be managed in different LSM-trees. First, from the namespace isolation perspective, namespace information is stored in the indexing structure along with the KV pair for index management. **Namespace Table (NSTable)** is an auxiliary memory component that stores the namespace information of the KV pair in the MemTable. As shown in Figure 5, NSTable keeps track of min and max keys per namespace and the total number of KV pairs per namespace. When KV pairs in the MemTable are made to an SSTable after a flushing operation, the namespace and NSTable information per namespace are also stored in the SSTable footer, where the meta region is stored. Accordingly, when reading a KV pair, the access is allowed only if the matching of both the key and its namespace is satisfied.

Cerberus employs the per-namespace dedicated LSM-tree index. In the per-namespace LSM-tree, the MemTable and the lowest-level SSTables (e.g., L_0) do not store KV pairs separately according to the namespace. They are mixed and stored in the MemTable and L_0 SSTables. On the other hand, in other levels (e.g., $L_1, \ldots, L_n$) of SSTables, KV pairs are segregated by their namespaces and indexed to different LSM-trees. This approach was designed for two reasons. First, the performance degradation caused by a mixture of data from any tenants in MemTable and L_0 SSTables is negligible. According to our experiments, KV pairs are rarely searched in MemTable and L_0 SSTables (less than 3.2%) compared to other level SSTables. The more the degree of multi-tenancy increases, the less KV pairs are searched in MemTable and L_0 SSTables. Second, the MemTable provisioning per tenant approach requires additional DRAM space. If the number of concurrent tenants increases, more DRAM capacity is required in proportion to the degree of multi-tenancy.

The main expected effect of the namespace-dedicated LSM-tree is that *ALKA* is reduced. The first reason is that KV pairs from different namespaces are indexed to different LSM-trees, preventing them from being indexed to the upper levels of LSM-trees. Thus, this mitigates the computation of MP_n by overlapping several times in the *ALKA* equation. In other words, the value of n becomes smaller. The second reason is that the number of BF loading from the NAND flash memory decreases during the retrieval of the KV pair. This corresponds to a decrease in the MP_n value itself in the *ALKA* equation.

4.3 Namespace Isolation Mechanism

Figure 6 illustrates a namespace isolation mechanism that segregates KV pairs from a global LSM-tree into per-namespace dedicated LSM-trees based on a tenant's namespace. Namespace isolation is performed during the background compaction process as in traditional LSM-trees. The compaction process reads the victim L_i and L_{i+1} SSTables from the NAND flash memory and merge-sorts them into a new L_{i+1} SSTable. Then, the new merged SSTable is written to the NAND flash memory. This isolation strategy does not require an additional SSTable read operation because it performs isolation for the victim SSTable already read during compaction. Here, the namespace of each entry in the victim SSTable is called the victim namespace.

When compaction for L_0 SSTable begins, Cerberus first chooses the victim SSTables and reads them into the DRAM memory (①). Since compaction is triggered at L_0, per-namespace isolation is enabled to classify KV pairs according to namespace. If compaction starts at a level other than L_0, the isolation task is disabled and performs the same as the traditional compaction flow. If isolation

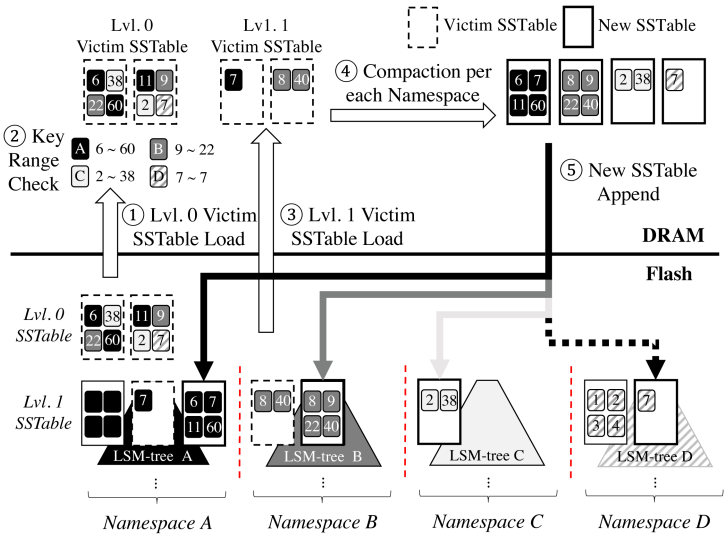


Fig. 6. Per-namespace isolation process.

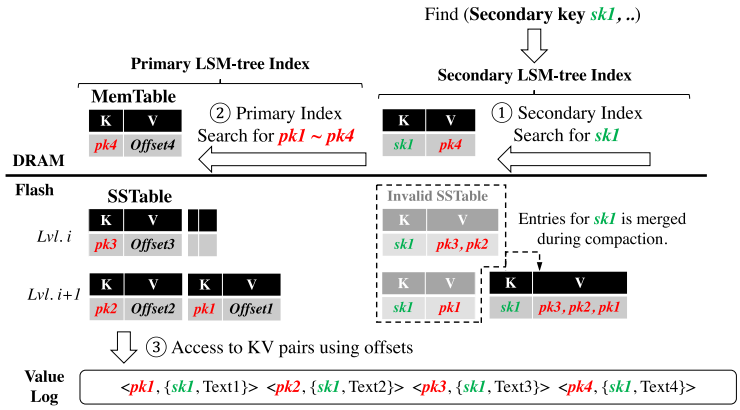
is enabled, the key range for each victim namespace in the L_0 victim SSTable is checked (②). This is done to verify whether the key range of the victim namespace is overlapped with the key range of the L_1 SSTable of the victim namespace LSM-tree. Then, L_1 SSTable with an overlapping key range is read into the DRAM memory (③). Next, for each entry in the L_0 victim SSTable, compaction is performed with the corresponding namespace's L_1 victim SSTable (④). This process creates a new SSTable for each victim namespace. Finally, the index entry of multiple namespaces that existed in L_0 victim SSTable is appended by indexing to L_1 SSTable of the per-namespace LSM-tree (⑤).

In the example shown in Figure 6, two L_0 victim SSTables are read during compaction. In this case, the L_1 SSTables of namespace A and B are read for isolation. For namespace A, the key range (7–7) of the second L_1 SSTable belongs to the key range (6–60) of the L_0 SSTable. For namespace B, the key range (9–22) of the L_0 SSTable belongs to the key range (8–40) of the first SSTable of L_1. A new L_1 SSTable is created by performing merge-sort for each namespace and is stored individually in the per-namespace LSM-tree.

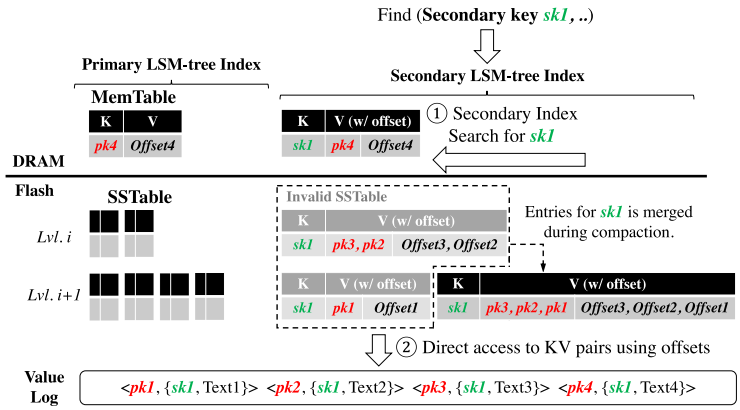
4.4 Value Addressable Secondary (VAS) Index

Note that the non-primary attribute and its corresponding value are stored as a JSON format as follows: *Value* $v = \{A_1 : value(A_1), \ldots, A_n : value(A_n)\}$, where the *value* ($A_i$) is a value for the non-primary attribute A_i. For an example of the tweet social networks dataset, tweet post can be a KV pair and *tweet ID* of this post can be a *pk*. The A_1 and A_2 can be *user ID* and *text body*, respectively, and *user ID* can be utilized as an *sk*. In order to process search query accompanied by *sk*-based search predicate, Cerberus maintains an *sk*-based secondary index inside a KVSSD.

4.4.1 State-of-the-art Secondary Index Design on KVSSD. Figure 7(a) shows that Cerberus adopted the state-of-the-art secondary indexing technique. This secondary indexing technique is currently used in popular LSM-tree-based NoSQL databases [1, 4, 37]. Cerberus also begins to construct a secondary index centered around specific non-primary attributes by using CreateIndex() API, which is described in Table 2. Similar to the primary index, data structures such as MemTable and SSTable also exist in the LSM-tree-based secondary index. When the KV pair is first written to the value log on the KVSSD, the *pk* and the value log offset are stored in the MemTable of the



(a) Traditional secondary indexing technique in KVSSD



(b) VAS indexing technique in KVSSD

Fig. 7. Comparison of secondary key-based search query processing using different secondary indexing techniques.

primary index. Next, specific *sk*, which is a non-primary attribute value, and the corresponding *pk* are also stored on the MemTable of the secondary index as a pair. For example, in the case of the tweet dataset, the *user ID* and *tweet ID* can be respectively stored as a key and value on the secondary index.

Later, the secondary index begins a background compaction to merge index entries with the same *sk* into a new index entry. Unlike the primary index compaction, the secondary index compaction does not delete the old version of the *sk* index entry in the process of merge. For instance, during the compaction of the secondary index in Figure 7(a), the *sk1* entries of L_i SSTable and L_{i+1} SSTable are simply merged, without removing any data.

As long as Cerberus maintains the *sk*-based secondary index, the following two types of *sk*-based search queries are possible: (1) the query that calls Get() API with the *sk* argument that finds and returns the *pk* list only and (2) the query that calls Find() API with the *sk* argument that checks the actual value for each found *pk* and returns a set of KV pairs. In Figure 7(a), if Find() is requested, Cerberus searches *sk1* entries upon secondary index (①). After the *s1* entries are found, the corresponding value, which is a *pk* list, can be identified. In this example, a total of four *pks* are

identified for *sk1*. Then, Cerberus internally requests primary index search for a found *pk* list to check the value log offset (②). Finally, Cerberus can read actual KV pairs by referencing the offsets stored in the primary index (③). If Get() with the *sk* is requested, Cerberus finds the *pk* list for an *sk*, returns it to the host, and terminates the process.

4.4.2 Proposed VAS Index Design on KVSSD. Figure 7(b) shows the proposed VAS index in Cerberus. The VAS index is equivalent to the traditional secondary index except that VAS keeps not only *pk* but also value log offset. At the time when a KV pair is first written to the value log, the value log offset is recorded in both the primary index and VAS index. While processing *sk*-based search query such as Get() with *sk* or Find(), Cerberus performs search to find the requested *sk* upon the VAS index (①). If the requested *sk1* is found in the VAS index, the corresponding *pk* list and value log offset are identified. If the query is Get() with the *sk*, Cerberus returns the *pk* list. On the other hand, if the query is Find(), Cerberus directly accesses the KV pair by referring to a value log offset stored in the VAS index (②). Therefore, Cerberus that adopts the VAS index does not require *pk*-based primary index search while processing an *sk*-based search query.

In particular, while Cerberus with the VAS index processes the Find(), the total size of the KV pairs to be transferred to the host can exceed the maximum host-KVSSD DMA transfer size (e.g., 1 MB). To address this case, Cerberus controls the $IsIter$ flag parameter in response to the Find() to inform the host that remaining result data exists. During the response process of Find(), if the size of the KV pair to be transferred by Cerberus exceeds the DMA threshold size, then Cerberus sets $IsIter$ and temporarily keeps track of the next KV pair to be transferred. After the user checks that $IsIter$ is set, then user calls Find() iteratively until $IsIter$ is unset. Then, Cerberus continues to transfer the remaining KV pairs. Finally, Cerberus completes the transfer by unsetting $IsIter$ when all KV pairs are transferred to the host.

The proposed VAS indexing technique leverages the following two characteristic of a modern KVSSD. First, the LSM-tree-based index adopts key-value separation [33]. Second, the KV pair offset referred by the storage engine of KVSSD is a logical block address. Thus, the proposed VAS index is not affected by the index entry update of the primary index and the changes of the physical address of the KV pair stored in the NAND flash memory. The main effect of the VAS index is that the total sum of latencies for each KV access ($\sum ALKA$) in Equation (2) can be reduced when Find() is processed. This is because when processing an *sk*-based search query that returns actual KV pairs, Cerberus searches the VAS index entry, bypasses the primary index search, and immediately accesses the KV pairs. Therefore, in Equation (2), the value of p is zero, and the value of $\sum ALKA$ is calculated only from $ALKA^{secondary}$.

4.5 In-storage Processing (ISP) with VAS Index

Traditionally, a query can be fused vastly with other queries [6, 40]. The query fusion allows multiple operations to be executed with a single call. For the tweet dataset as an example, the user may demand on a fused analytic query in order to read the tweet posts written by a specific *user ID* and to sort the posts by last modification date. The query fusion is beneficial in that it is possible to skip the process of transferring and storing the intermediate result of each query to the storage. When the certain fused analytic query contains a compute operation, Cerberus performs in-storage processing using SSD's internal CPU and DRAM resources. The AnalyticOp() API described in Table 2 is a use-case of *sk*-based search query fused with sort operation. If this API is requested to Cerberus, then Cerberus accesses KV pairs directly using the VAS index and sorts using the SSD's internal resources. After that, Cerberus can return only the top-K (e.g., K = 50) outputs to the user in the case of AnalyticOp(). As a result, Cerberus expedites such analytic query processing by eliminating transmission overhead of the intermediate result of the search query.

Table 2. Cerberus API for Namespace and Performance Isolation with Secondary Indexing Support

Cerberus API Routine	Routine Description
ns, *idx_id* = **Create**(*attr*)	Create and initialize the new primary LSM-tree based on specific attribute (*attr*) and return the index ID (*idx_id*) and specific namespace (*ns*) to user.
Destroy(*ns*)	Clear contents of the LSM-trees corresponding to namespace (*ns*) and destroy the KV database of the namespace (*ns*).
idx_id = **CreateIndex**(*ns*, *attr*)	Create and initialize the new secondary LSM-tree for specific attribute (*attr*) and return the secondary index ID (*idx_id*) in the namespace (*ns*).
DropIndex(*ns*, *idx_id*)	Clear contents of the secondary LSM-tree (*idx_id*) corresponding to namespace (*ns*) and destroy the secondary index.
Put(*ns*, *pk*, *v*)	Store the value (*v*) of the primary key (*pk*) to a primary LSM-tree and synchronously update to a secondary LSM-tree corresponding to the namespace (*ns*).
v or *pklist* = **Get**(*ns*, *pk* or *sk*)	Retrieve the value (*v*) of the primary key (*pk*) from the primary LSM-tree or retrieve the *pk* list (*pklist*) associated with the secondary key (*sk*) from the secondary LSM-tree corresponding to the namespace (*ns*).
Delete(*ns*, *pk*)	Delete the value (*v*) of the primary key (*pk*) from the primary LSM-tree corresponding to the namespace (*ns*).
vlist = **Find**(*ns*, *sk*, *IsIter*=False)	Retrieve the value list (*vlist*) of the secondary key (*sk*) from the secondary LSM-tree index corresponding to the namespace (*ns*). The *IsIter* indicates whether the query should be repeated.
pklist = **AnalyticOp**(*ns*, *sk*, *comp*, *k*, *attr*)	Perform the following operations in our experiment: (1) Find the candidate of primary key list (*pklist*) associated with the secondary key (*sk*) from the secondary index corresponding to the namespace (*ns*), (2) find the actual value of each *pk* candidate from the value log, (3) sort the *pks* based on the specified attribute (*attr*) in ascending or descending order (*comp*), and (4) return only *pklist*s belonging to top-K (*k*) to the host.

4.6 Storage Space Management in Cerberus

Cerberus operates the key-value storage engine by utilizing the logical and physical addressing system and its translation logic provided by the SSD **Flash Translation Layer (FTL)**. Specifically, when storing KV pair in Cerberus, the physical page to be used is obtained through the existing physical page allocator inside the SSD. This physical address is then mapped to an available logical address from the logical linear space. The mapping information is managed by the FTL. In Cerberus, the storage space is not differentiated according to the namespace when allocating physical pages. When Cerberus reads the KV pair, the logical address recognized by the engine is translated to a physical address through FTL and the KV pair is retrieved by referring to the physical address.

In order to reclaim the space, Cerberus also supports tenant deletion as well as deletion of the KV pair. When the Destroy() API described in Table 2 is requested, Cerberus begins to reclaim the page occupied by the tenant's primary and secondary index and KV pairs. Cerberus first removes MemTable entries corresponding to the namespace of the target tenant. Next, Cerberus accesses the SSTable without the lowest level (e.g., L_0) to obtain the value log offfsets of each KV pair. Cerberus then invalidates all disk pages on which tenant KV pairs are stored. Cerberus also invalidates the page on which the SSTable of the LSM-tree index is stored. Later, these invalidated pages are reclaimed through **garbage collection (GC)** of SSD. In particular, the index entries of the L_0 SSTable are not immediately invalidated because entries of other namespaces are mixed at L_0. If Cerberus immediately invalidates pages containing L_0 entries, then not only the entries of the target tenant but also the entries of the other tenant will be invalidated together. Thus, it

Table 3. Cosmos+ OpenSSD Platform Specification

Component	Specification
SoC	Xilinx Zynz-7000
	ARM Cortex-A9 (up to 1,000MHz)
	HYU Tiger 4 Controller (NVMe Controller in FPGA)
NAND Module	1TB module, NVDDR2
	4 Channel, 8 Way
	18,048B Page (1664B Spare)
Interconnect	PCIe Gen2 8-lane
FTL	Page-level Mapping, On-demand GC

Table 4. Host Machine Specification

Component	Specification
CPU	Intel Core i7-10700 @ 2.9GHz 8C/16T
RAM	16 GB @ 2,666MHz
OS	Linux Kernel 5.8.0

eventually forces Cerberus to perform out-of-place updates to other SSD pages so that the entries of the other tenant will not be deleted. To avoid additional SSD page writes, Cerberus exceptionally delays the L_0 SSTable invalidation. Instead, the target tenant's L_0 SSTable index entries are just left and not segregated into L_1 during Cerberus's per-namespace isolation process. The per-namespace isolation of Cerberus reads the L_0 SSTable at the time of the compaction, creates L_1 SSTables of different LSM-trees depending on the namespace of each entry, and invalidates the existing L_0 SSTable. Accordingly, index entries of the target tenant left without being segregated are naturally invalidated. Cerberus only requests the invalidation of pages where the actual target tenant's KV pairs are stored through the index entry of the L_0 SSTable to be deleted. Then, the entire invalidation process for the L_0 SSTable is completed.

4.7 Cerberus API Routines

Each user can send KV requests to Cerberus via the key-value API routines in a host. The major key-value API routines are presented in Table 2. For example, the Get() API in Table 2 is an overloaded function that returns different outputs according to the input type similarly with KV-Stores. The AnalyticOp() API is to request a query in which *sk*-based search and compute operations (e.g., sort) are fused. By using this, we will evaluate the performance of Cerberus even when the search query is fused with other in-storage analytic operations. The key-value API uses a system call (e.g., ioctl) to pass KV pairs passed from the user to the KVSSD kernel driver. Then, the kernel driver writes the namespace in the NSID region and key in the LBA region of the NVMe command. The parameter *IsIter* of Find() is stored in the unused field of the NVMe command. The memory address where the value is stored is recorded in the page list region of the NVMe command. Then, the NVMe command is sent to Cerberus.

5 EVALUATION

5.1 Experimental Setup

We prototyped iLSM-SSD [31] and Cerberus on the Cosmos+ OpenSSD platform. Table 3 shows the details of the Cosmos+ OpenSSD hardware platform. The Cosmos+ OpenSSD is equipped with Xilinx Zynq-7000, which has an ARM Cortex-A9 processor with two cores running at 1GHz, FPGA, and 1GB DDR3 DRAM. The FPGA operates the SSD controller including NAND Flash controllers, NVMe controllers, and PCIe controllers. The page-level FTL and garbage collection are running at CPU. Cosmos+ OpenSSD is connected to the host machine via PCIe Gen2 8-lane with NVMe protocol. The specification of host machine is described in Table 4. All experiments are initiated from the host machine. We compare the following three systems:

- KVSSD-baseline: It refers to existing KVSSD such as iLSM-SSD [31] that only supports primary key-based index. It does not support namespace and performance isolation and secondary index for *sk*-based search query.

Table 5. Secondary Key-based Search Queries to Measure the Query Performance

	Dataset	No.	**Description** (the non-primary attributes are highlighted in **bold**)	**Secondary Keys**	**Return Type**
Search Query	Glacier	Q1	List top 50 of IDs of glacier that belong to specified **Political Unit**	FRAN, BHUT, AUST, BOLI	*pklist*
		Q2	List top 50 of **All information** of glacier that belongs to specified **Political Unit**	FRAN, BHUT, AUST, BOLI	*vlist*
		Q3	List top 50 of **Basin code** of glacier that belongs to specified **Political Unit**	FRAN, BHUT, AUST, BOLI	*vlist*
	Tweet	Q4	List ID of tweet post that belongs to specified **User ID**	2517, 3161, 1153	*pklist*
		Q5	List **All information** of tweet post that belongs to specified **User ID**	2517, 3161, 1153	*vlist*
		Q6	List **Text body** of tweet post that belongs to specified **User ID**	2517, 3161, 1153	*vlist*
Analytic Query	Glacier	Q7	Sort IDs of glacier by the height in the specified **Political Unit** and list top 50 of IDs	FRAN	*pklist*

The attributes refer to specific non-primary attribute used as a secondary key.

- Cerberus: It refers to a KVSSD with per-namespace dedicated LSM-tree for namespace and performance isolation. It also supports state-of-the-art secondary index for *sk*-based search query [37].
- Cerberus-VAS: It refers to Cerberus with VAS index. It enables KVSSD to access the value log directly, bypassing primary index search in case of *sk*-based query.

Workloads: We used a synthetic KV dataset for per-namespace LSM-tree index evaluation, where the size of each KV pair is 4B and 1KB. During the experiment for the per-namespace LSM-tree index, the number of concurrent users issuing KV requests, or degree of multi-tenancy, was increased from one to eight, and key was randomly specified. Each user issued the same number of 1 million KV requests for each Put() only and Get() only (with primary key) workloads.

To evaluate the VAS indexing technique, we used publicly available scientific datasets such as glaciers [43] and social data such as Twitter's tweet [47]. For the glacier dataset, the total number of KV pairs was 104.6K with the key size 8B and the value size 0.963KB on average. For the tweet dataset, the total number of KV pairs was 132.8K with the key size 8B, and the value size approximately 8KB on average. Table 5 lists the set of queries used to evaluate the VAS index based on the datasets described above. The glacier dataset is used for Q1~Q3 and the tweet dataset is used for Q4~Q6. In particular, Q1 and Q4 use Get() (with secondary key), which returned the *pk* list as a result. On the other hand, Q2, Q3, Q5, and Q6 use Find(), which return the value list as a result. The Q7 is an analytic query, which is used for evaluating not only search operation but also in-storage computation of Cerberus.

5.2 Per-tenant Performance Comparison

We measured both per-tenant throughput and response time of Cerberus and KVSSD-baseline for Put() only and Get() only (with primary key) workloads. Figure 8(a) shows the negligible throughput difference between KVSSD-baseline and Cerberus for a Put()-only workload. The throughput overhead of Cerberus lies within 1% compared to KVSSD-baseline regardless of the number of concurrent tenants. This slight overhead with Cerberus is due to the namespace isolation. Specifically, the isolation overhead is attributed to reading the L_1 SSTable of the LSM-tree corresponding victim namespace. Also, Figure 8(b) shows little response time difference between the KVSSD-baseline and Cerberus. In particular, Cerberus shows only 4% additional latency compared to baseline, when the number of tenant is eight.

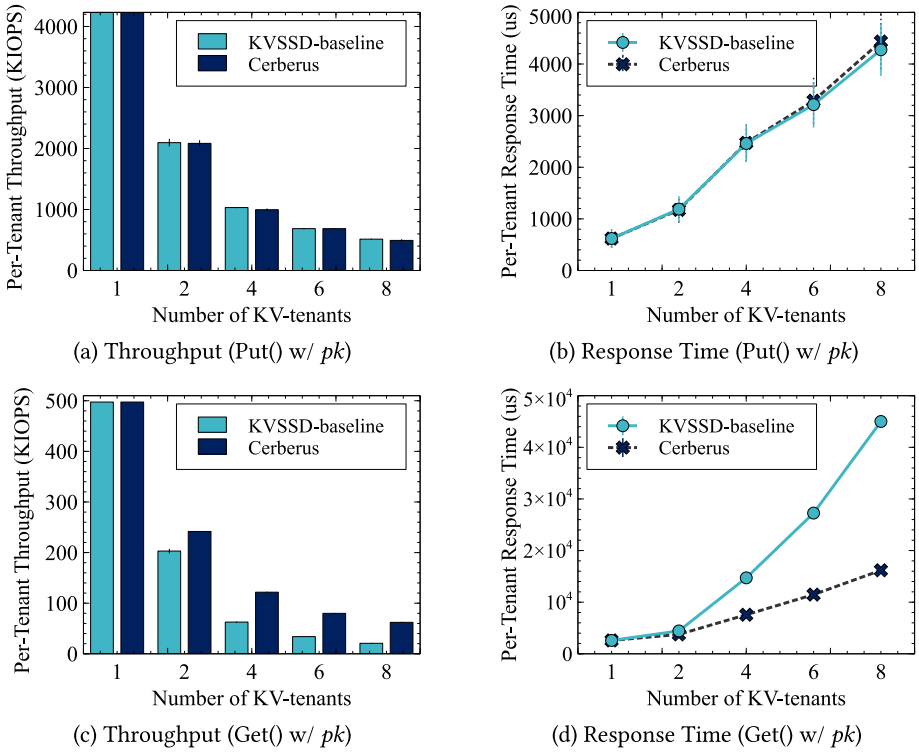


Fig. 8. Throughput and average response time of KVSSD-baseline and Cerberus with KV-tenants (1–8).

In Figure 8(c), the Get() throughput difference between KVSSD-baseline and Cerberus is prominent. When the number of concurrent tenant is two, four, six, or eight, Cerberus shows 1.1, 1.9, 2.3, and 2.9× higher Get() throughput than KVSSD-baseline, respectively. This is because Cerberus adopts the per-namespace LSM-tree. Therefore, Cerberus reduces the depth of LSM-tree and reduces the number of BF loads required during the KV pair search process. Figure 8(d) shows a comparison of response times. The difference in average response time between KVSSD-baseline and Cerberus becomes evident as the number of tenants increases. Specifically, if the number of tenants is eight, Cerberus has a 2.78× lower average response time than KVSSD-baseline. We additionally measured the Get() response time when the number of tenants is 16. We observed the response time of KVSSD-baseline becomes 229.3 ms, which is an increase of 100 order of magnitude compared to the case when the number of tenants is one. On the other hand, the response time of Cerberus becomes 40.1 ms, which is an increase of 10 order of magnitude compared to the case when the number of tenants is one. The performance of Cerberus-VAS is the same as that of Cerberus while processing the *pk*-based Get() request. This is because both Cerberus and Cerberus-VAS handle the request by only consulting the primary index. On the other hand, the Put() performance of Cerberus-VAS is slightly degraded compared to Cerberus. This is because the overhead for creating the secondary index is included in the Put() process. We reported the performance degradation of Cerberus-VAS in Section 5.4. Besides, we further conducted experiments on workload, which is mixed with Put() and Get() operations, such as YCSB [10] workload patterns. In the workload, Put() and Get() operations are equally mixed. According to the result of the experiment, it was confirmed that the per-tenant throughput in changing the number of tenants is hardly different from that of Get()-only workloads. For instance, when

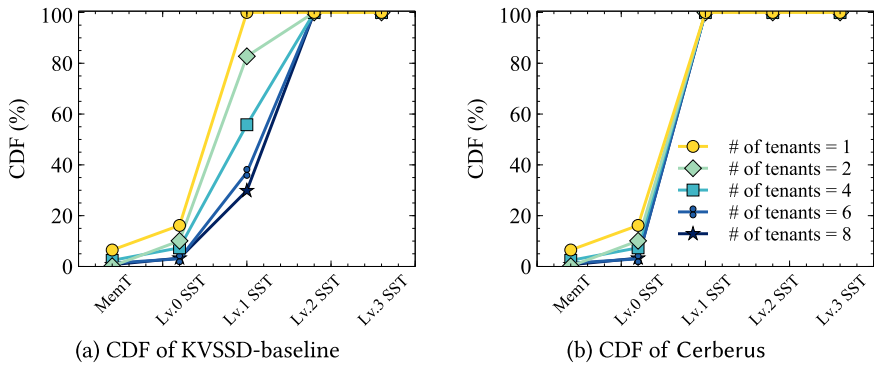


Fig. 9. Level distribution of where KV data is indexed in the primary LSM-tree index.

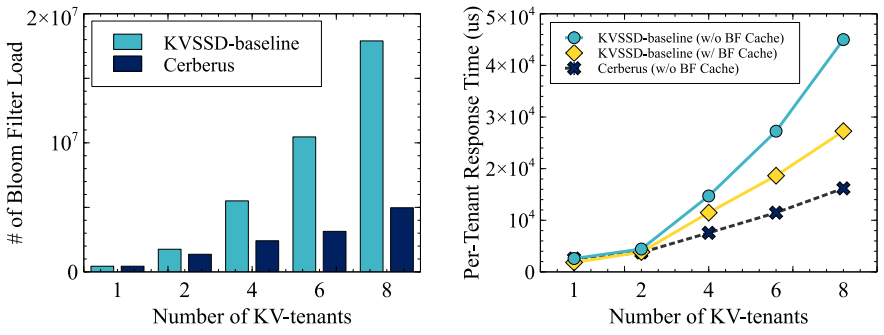
the number of tenants is four, the per-tenant throughput of Cerberus improves by 1.53× compared to KVSSD-baseline. This result shows that the read performance benefit from Cerberus's per-namespace LSM-tree is still significant while the write overhead of Cerberus is negligible.

5.3 Impact of Per-namespace LSM-tree

Figure 9 represents a CDF on which the level of the LSM-tree index entries are searched during Get(). Figure 9(a) is a CDF of the KVSSD-baseline. When the number of tenants is one, the search is completed only with the index entries of MemTable, L_0, and L_1 SSTable. However, as the number of tenants increases, the number of searches from the lower-level index of the LSM-tree decreases and Get() is processed, reaching the higher-level index. This is due to the fact that all index entries of tenants are managed by a global-shared LSM-tree, thus forming a deeper level of the LSM-tree. Specifically, as the number of tenants increases from one to eight, the percentage at which the KV pair is searched in L_1 is reduced from 83.8% to 26.55%, and the percentage at which the KV pair is searched in L_2 is increased from 0% to 70.1%. The percentage of KV pairs searched in MemTable and L_0 SSTable is only 3.2%. On the other hand, Figure 9(b) is a CDF of Cerberus. Since KV pairs from other namespaces are indexed to individual LSM-trees, search process is completed with only index entries of MemTable, L_0, and L_1 SSTable regardless of the number of tenants. These results are evidence that a per-namespace LSM-tree can lower the number of recursions in *ALKA* Equation (1).

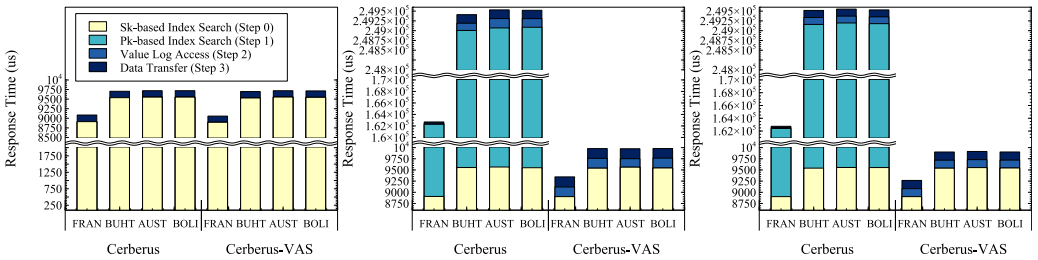
Next, Figure 10(a) represents how many BF loads are performed during Get(). Cerberus can reduce the number of BF loads with the help of the per-namespace LSM-tree. In particular, when the number of tenants is eight, Cerberus results in 3.6× fewer BF loads than the KVSSD-baseline. These results are evidence that MP_n in *ALKA* Equation (1) can be reduced. This BF overhead can be further minimized by caching if there is enough DRAM inside the SSD.

Effectiveness of Caching BFs: Originally, BF was designed as a memory-resident data structure. Thus, the BF overhead can be further minimized by caching if there is enough DRAM inside the SSD. In order to see the impact of caching BF and compare it with the effect of per-namespace LSM-tree in Cerberus, we implemented KVSSD-baseline, which is able to cache all of BF's of tenants. We measured the average response time of BF cache-disabled KVSSD-baseline, BF cache-enabled KVSSD-baseline, and BF cache-disabled Cerberus. Figure 10(b) shows the comparison of the average response time for these three cases during Get(). When the number of tenants is one, the KVSSD-baseline with BF cache (w/ BF cache) improves the response time by 29.1% over the KVSSD-baseline without BF cache (w/o BF cache). This is trivial in that caching BF reduces the number of NAND flash memory access. When the number of tenants is eight, the



(a) Bloom Filter load (b) Impact of Bloom Filter cache

Fig. 10. Change of the number of Bloom Filter (BF) load and impact of caching BF with KV-tenants (1–8).



(a) Q1 (Get() w/ *sk*) (b) Q2 (Find() w/ *sk*) (c) Q3 (Find() w/ *sk*)

Fig. 11. Query response time for processing Q1–Q3 in Table 5.

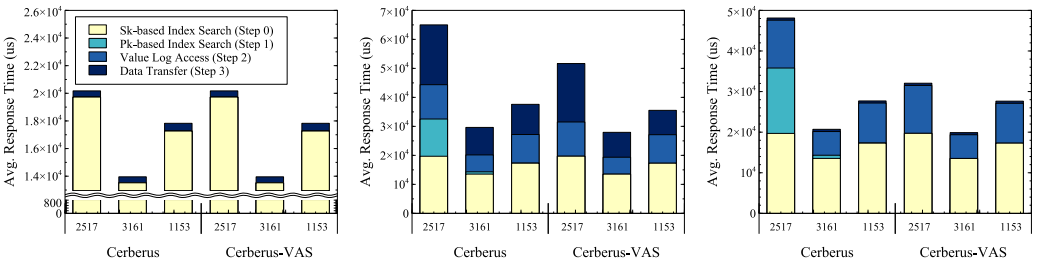
KVSSD-baseline w/ BF cache still improves the response time by 17% over the KVSSD-baseline w/o BF cache. However, as the number of tenants increases, we observe that the KVSSD-baseline w/ BF cache cannot outperform the Cerberus w/o BF cache. For example, when the number of tenants is eight, Cerberus w/o BF cache shows 2.2× lower average response time than the KVSSD-baseline w/ BF cache. This implies that per-namespace LSM-tree that reduces the depth of index has more impact on Get() performance than caching BF.

5.4 Evaluation for Effectiveness of VAS Index

The performance of each query on Table 5 is analyzed with the time breakdown, measuring the following four steps: first, *sk*-based secondary index search time to read the *pk* list (Step 0); second, the *pk*-based primary index search time to read value log offset (Step 1); third, the NAND flash memory read time to read actual KV pair (Step 2); and fourth, KV pair transfer time to host (Step 3).

We first measured the average response time of Cerberus and Cerberus-VAS for Q1, Q2, and Q3 on the glacier dataset. Q1, Q2, and Q3 are top-50 queries that only return 50 *pk* lists or 50 KV pairs. Figure 11 shows the average response time of each query processing with different *sks* (*FRAN*, *BUHT*, *AUST*, and *BOLI*). Figure 11(a) shows the response time of Q1 processing using Get() with these *sks*. In Figure 11(a), the response time of Cerberus and Cerberus-VAS is almost the same. This is because both systems perform *sk*-based index search to read the *pk* list corresponding to the requested *sk* (Step 0) and then return the *pk* list to the host (Step 3).

On the other hand, Figures 11(b) and 11(c) show that Cerberus-VAS significantly outperforms Cerberus in terms of average response time when the Find() is requested. For a specific example,



(a) Q4 (Get() w/ *sk*) (b) Q5 (Find() w/ *sk*) (c) Q6 (Find() w/ *sk*)

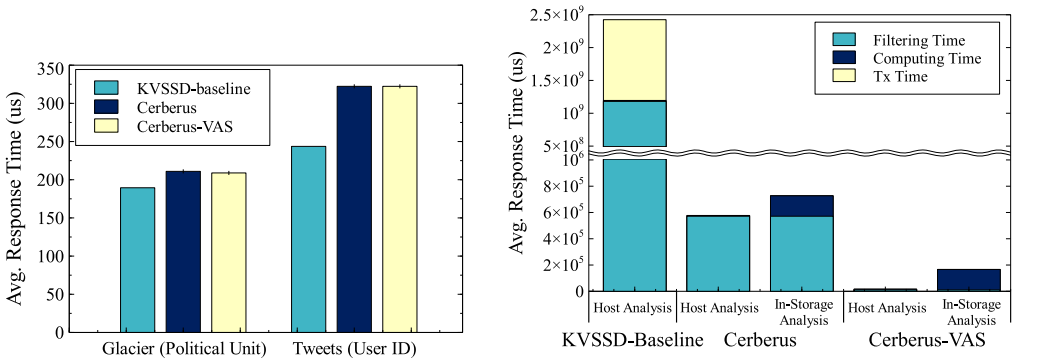
Fig. 12. Query response time for processing Q4–Q6 in Table 5.

Cerberus-VAS shows 9.31× lower response time than Cerberus when *sk* is *AUST* during Q2 processing. Cerberus uses *a sk*-based secondary index to find the *pk* list whose *Political Unit* attribute value is *AUST* (Step 0). Then, Cerberus checks the value log offset through *pk*-based primary index search (Step 1) and reads actual KV pairs (Step 2). During this process, *pk*-based index search time accounts for 84% of the total response time. On the other hand, Cerberus-VAS can directly check the *pk* list and value log through VAS index (Step 0) and read KV pairs (Step 2). Thus, the *pk*-based index search (Step 1) is unnecessary. Meanwhile, the response time of the case where the *Political Unit* value is *FRAN* has 1.4× lower response time than other cases in Cerberus. This is because during *pk*-based index search, the *pk* list corresponding to *FRAN* is always searched in L_1 of the tree. On the other hand, the *pk* list corresponding to other attribute values (*BUHT*, *AUST*, and *BOLI*) is always searched in L_2 of the tree.

We also measured average response times of Cerberus and Cerberus-VAS for Q4, Q5, and Q6 on the tweet dataset, which has more large-sized KV pairs than the glacier dataset. Figure 12 shows average response time of each *sk*-based query processing with secondary keys (*2,517*, *3,161*, and *1,153*). Figure 12(a) exhibits the response time of Q4 processing using Get(). While Q4 is processed, only Steps 0 and 3 are required, which is the same as Q1 processing. However, in Figure 12(a), Q4 processing has 5.89× higher average response time than Q1 processing. This is attributed to *sk*-based secondary index search time. During Q4 processing, the rate at which *sk* is searched at L_1 is close to 88% in the *sk*-based index, while top-50 Q1 processing does not reach to L_1 during *sk*-based index search.

Figures 12(b) and 12(c) show that Cerberus-VAS has still lower response times than Cerberus. However, Cerberus-VAS only gives 1.25× improvement in response time at most during Q5 processing on the tweet dataset, which is less than the performance improvement of Q2 processing on the glacier dataset. This is because *pk*-based primary index search during Q5 processing on the tweet dataset takes 11.6× less time than *pk*-based index search during Q2 on the glacier dataset. Unlike *pk*-based index search of Q2 processing, 87% of *pk* is searched at MemTable on average during pk-based index search of Q5 processing. In particular, Figure 12(c) shows that Q6 processing has lower response times than that of Q5 processing. This is because the total amount of KV pairs transferred to the host during Q6 processing is 29.9× less than that of Q5 processing. While Q5 returns the entire KV pair, which is 8KB in size on average, Q6 returns only text body of tweet, which is 0.5KB on average.

To check if the effectiveness of the proposed Cerberus-VAS index is valid regardless of the number of KV pairs, we increased the small glacier dataset by a factor of 10 to create a large dataset and requested Q1, Q2, and Q3 using the same secondary keys. The results showed that for both Cerberus and Cerberus-VAS, the *sk*-based index search time increased for the large dataset compared to the small dataset, resulting in a 2.04× to 2.29× increase in response time. This is due to



(a) Put() w/ *pk* and *sk*

(b) AnalyticOp() w/ *sk* (Search + Compute operation)

Fig. 13. Comparison of the average response times of KVSSD-baseline, Cerberus, and Cerberus-VAS for processing Put() (with *pk* and *sk*) and AnalyticOp() (with *sk*). It is possible for Cerberus and Cerberus-VAS to enable both host-driven and SSD-driven analytic approaches during AnalyticOp() because of a secondary index.

a larger number of KV pairs being indexed to the LSM-tree, leading to a longer LSM-tree search time.

In the case of Q2 and Q3, we confirmed that the data transfer time for Q2 and Q3 remained unchanged for both small and large datasets in both Cerberus and Cerberus-VAS. For a large dataset, Cerberus took up to 2.3× longer for *sk*-based and 9.01× longer for *pk*-based index search compared to the small dataset, respectively. Meanwhile, Cerberus-VAS took up to 2.28× longer for *sk*-based index search in a large dataset. Unlike Cerberus, Cerberus-VAS does not always perform *pk*-based index searches as it directly accesses the value log. The Q3 results were similar to the Q2 results. Our experimental results indicate that the effectiveness of the Cerberus-VAS index is still valid.

Time Cost of Constructing Secondary Index: Figure 13(a) shows the average response time of KVSSD-baseline, Cerberus, and Cerberus-VAS for Put() workload on the glacier and tweet dataset. The response time of Cerberus is 11% and 32% higher than that of KVSSD-baseline when the glacier and tweet dataset are used, respectively. This additional overhead is attributed to constructing an *sk*-based secondary index. While constructing the secondary index, the *sk* and corresponding *pk* pair is written into MemTable and the compaction is also periodically triggered at the secondary index. These additional operations result in performance overhead during Put() workload. Meanwhile, the response time of Cerberus-VAS is almost identical to that of Cerberus. There is only a 1.4% difference at most between Cerberus and Cerberus-VAS. In Cerberus-VAS, the value log offset is additionally written to the index entry of the secondary index, which is negligible to performance of Put().

Space Cost of Maintaining Secondary Index: Unlike KVSSD-Baseline, Cerberus and Cerberus-VAS require additional flash memory space for storing the secondary index. For example, in the case of the glacier dataset, KVSSD-Baseline occupies 264 pages for storing the primary index and exhibits about 4.22 MB of flash memory space overhead. On the other hand, Cerberus and Cerberus-VAS need to store secondary indexes additionally, so a total of 591 pages are used and flash memory space overhead of 9.45 MB is shown. It is enough to cover this overhead for our 1 TB of SSDs. We observed that the space overhead of Cerberus is the same as that of Cerberus-VAS in terms of the number of pages used in KVSSD. This is because Cerberus-VAS writes data in SSD page unit, and even if the value log offset (4 B) and namespace information (1 B) are additionally stored per index entry, it does not exceed SSD page size (e.g., 16 KB). Assuming the number of

entries in the secondary index is fixed, as the entry size increases, it is possible to exceed the SSD page size. Overall, the space amplification of Cerberus-VAS increases in proportion to how many additional SSD pages are required. For instance, if there are 16 KB SSD pages and 1,000 index entries of 16 B entry size, then one more page is additionally required.

5.5 Evaluation for In-storage Processing of Cerberus

Figure 13(b) shows the average response time of different analytical query processing approaches on the glacier dataset. The analysis task is to sort KV pairs by a specific *sk*, which is a CPU-intensive computation. Then, after returning only the top-50 KV pairs, the task is completed. For a KVSSD-baseline, Get() (with *pk*) is used for reading all KV pairs because it does not have an *sk*-based secondary index. In this case, only host-side analysis is a feasible approach. In Figure 13(b), the host analysis with the KVSSD-baseline took about 40 minutes, which is the worst latency among comparative groups. The filtering time of the KVSSD-baseline is a time of *pk*-based primary index search, which is a leading factor. An AnalyticOp() API, described in Table 2, is used in Cerberus and Cerberus-VAS because the secondary index exists. The host-analysis approach on Cerberus shows 577.3 ms. It is confirmed that the secondary index is effective to significantly shorten the time of *sk*-based search query processing. However, the in-storage analysis on Cerberus shows 727.24 ms, which is 1.25× longer than that of the host-analysis approach. Since the KVSSD has a relatively wimpy ARM-based processor, unlike the host, the time for sort (computing time) increases by up to 29×. On the other hand, the in-storage analysis on Cerberus reduces the KV pair transmission time (Tx Time) by up to 3.9× compared to that of host analysis with Cerberus. This is because the in-storage analysis approach returns only the top-50 KV pairs as a result after sorting inside the KVSSD. In Cerberus-VAS, both the host-analysis and in-storage analysis show improvement on response time compared to Cerberus. Specifically, the filtering time of Cerberus-VAS is reduced by 53.3× compared to that of Cerberus because Cerberus-VAS bypasses the *pk*-based primary index during KV pair search.

6 RELATED WORK

Key-value SSDs. Several previous studies have proposed the implementations of KV-Store inside SSDs [7, 12, 20, 22, 31, 46]. The aforementioned KVSSDs primarily focus on designing primary key-based indexes and processing searches based on them. LSM tree-based KVSSDs such as iLSM-SSD [31] and PinK [20] support primary key-based queries. KAML [22] uses a hash-based index data structure for its internal index implementation. KAML supports namespace isolation, but KAML does not provide indexes based on secondary keys. These KVSSDs have adopted several techniques to optimize primary key-based query services: iLSM-SSD [31] adopts key-value separation techniques to overcome internal resource constraints. PinK [20] manages to minimize tail latency by pinning low-level SSTables at DRAM. However, no previous study looked at optimizing performance with namespace isolation in the LSM tree-based KVSSD. Other than that, resource (CPU and Memory) contention between different tenants is one of the important issues in a multi-tenant environment. To address such issues, there have been several studies on optimizing SSD for multi-tenancy use, such as SSD-level **Quality of Services (QoS)** support [26], data placement exploiting multilevel parallelism of SSD [19], and fair multi-resource allocation on the flash-based system [9]. However, these are out of the scope of our research. Instead, in order to release the significant performance bottleneck issue on the LSM-tree, we focus on isolating tenants' index in KVSSD, which is independent of SSD optimization works. Our future work is to build a KVSSD that integrates both our proposed design and the aforementioned studies.

In-Storage Processing. ISP uses the SSD's internal hardware resources such as CPU and memory for out-of-core execution inside the SSD [16, 23, 24, 27, 28, 32, 39, 44, 45, 48]. The ISP not only

frees up CPU and memory resources on the host but also reduces the cost of moving data between the host and the device.

Specifically, several researchers investigated accelerating DB applications with ISP on SSDs. FCAccel [44] integrated a column-oriented field-programmable gate-array-based acceleration engine into an SSD to offload SQL operators to the SSD. Aquoman [48] is an SSD with a general analytic query processor prototyped in an FPGA for ISP on the SSD. There was also a study on running table scan workloads on SSDs in POLARDB [45]. Although there have been some efforts to process DB queries on SSDs in hardware, previous studies have not looked at the problem of processing complex search queries in software in the LSM-based KVSSD.

Key-value Searches Using Secondary Indexes. KV-Stores have become a backbone of large-scale modern applications [13, 15]. Such modern applications including scientific applications oftentimes tend to access or query data based on several attributes different from primary key-based indexes [25]. Therefore, several existing works on KV-Stores maintain standalone secondary indexes to support non-primary attribute-based search queries. For example, a scientific dataset file such as HDF5 [14] contains thousands of secondary attributes, which are queried for various processing and analytics [25, 50]. To meet such needs, a recent work proposed **Group-Split-Merge (GSM)** [25] atop PMEMKV to support queries spanning over several attributes for scientific data formats such as HDF5 [14]. MIQS [51] is another work that offers secondary key queries for scientific datasets by exploiting a set of adaptive Radix trees and self-balanced search trees. MongoDB [4] follows the traditional relational databases approach and maintains a B-tree-based secondary index to support secondary search queries. Similarly, BigTable [8] and Cassandra [1] adopt LSM-tree-based secondary indexes to entertain secondary key-based queries. However, both MongoDB [4] and BigTable [8] perform eager updates on secondary indexes, requiring index entry update for every write operation, resulting in performance degradation. On the contrary, Cassandra [1] performs lazy updates on the secondary index and executes compaction in the background to mitigate the performance overhead. Note that all these studies build secondary indexes on host-based KV-Stores and they are not designed for KVSSDs.

7 CONCLUSION

This article proposes Cerberus, a design abstraction to enable namespace and performance isolation for LSM-tree-based KVSSDs in a multi-tenant environment. The key idea is to assign each user a dedicated namespace, ensuring user data isolation in the presence of multiple users and constructing a per-namespace-based LSM-tree to meet the performance guarantees. Furthermore, Cerberus is able to build and maintain secondary indexes to process non-primary attribute-based search and query requests. Specifically, Cerberus can streamline the process of the search queries by using VAS indexes. The key idea is to keep the address of the KV pair in the secondary index entries to access directly without a primary index search. Finally, Cerberus is able to process and accelerate the analytic query processing that combined search with computing operations. We prototyped Cerberus on the FPGA-based Cosmos+ OpenSSD in a Linux environment. The experiment and evaluation confirm that Cerberus restricts tenants' access to the corresponding namespaces and outperforms the traditional KVSSD in terms of performance. The experimental results also show that the VAS index of Cerberus gains huge performance improvement for various non-primary attribute-based search query scenarios compared to the traditional secondary index. According to the ISP experiment, Cerberus also outperforms the host side in terms of the analytic query processing performance.

REFERENCES

[1] 2013. Cassandra. http://cassandra.apache.org.
[2] 2017. Cosmos+ OpenSSD Platform. http://www.openssd-project.org/.

[3] 2018. EleasticSearch. https://www.elastic.co/.
[4] 2020. MongoDB Manual. https://docs.mongodb.com/manual/.
[5] Stefan Aulbach, Torsten Grust, Dean Jacobs, Alfons Kemper, and Jan Rittinger. 2008. Multi-tenant databases for software as a service: Schema-mapping techniques. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. 1195–1206.
[6] Nicholas J. Belkin, Colleen Cool, W. Bruce Croft, and James P. Callan. 1993. The effect multiple query representations on information retrieval system performance. In *Proceedings of the 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. 339–346.
[7] Janki Bhimani, Jingpei Yang, Ningfang Mi, Changho Choi, and Manoj Saha. 2021. Fine-grained control of concurrency within KV-SSDs. In *Proceedings of the 14th ACM International System and Storage Conference (SYSTOR'21)*. ACM, 1–12.
[8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems* 26, 2 (2008), 1–26.
[9] Wonil Choi, Bhuvan Urgaonkar, Mahmut Taylan Kandemir, and George Kesidis. 2022. Multi-resource fair allocation for consolidated flash-based caching systems. In *Proceedings of the 23rd Conference on 23rd ACM/IFIP International Middleware Conference*. 202–215.
[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. 143–154.
[11] Peter C. Dillinger and Stefan Walzer. 2021. Ribbon filter: Practically smaller than bloom and xor. *arXiv preprint arXiv:2103.02515* (2021).
[12] Samsung Electronics. 2018. Samsung Smart SSD. https://samsungatfirst.com/smartssd-ocp/.
[13] Facebook. 2017. LevelDB. https://github.com/google/leveldb.
[14] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. 2011. An overview of the HDF5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases (AD'11)*. 36–47.
[15] Google. 2012. RocksDB: A Persistent Key-value Store for Fast Storage Environment. https://rocksdb.org.
[16] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. 2016. Biscuit: A framework for near-data processing of big data workloads. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*. 153–165.
[17] Ajay Gulati, Arif Merchant, and Peter J. Varman. 2007. pClock: An arrival curve based approach for QoS guarantees in shared storage systems. *ACM SIGMETRICS Performance Evaluation Review* 35, 1 (2007), 13–24.
[18] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture: A Quantitative Approach*. Elsevier.
[19] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Chao Ren. 2012. Exploring and exploiting the multilevel parallelism inside SSDs for improved performance and endurance. *IEEE Trans. Comput.* 62, 6 (2012), 1141–1155.
[20] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. 2020. PinK: High-speed in-storage key-value store with bounded tails. In *Proceedings of the USENIX Annual Technical Conference (ATC'20)*. USENIX, 173–187.
[21] Shvetank Jain, Fareha Shafique, Vladan Djeric, and Ashvin Goel. 2008. Application-level isolation and recovery with solitude. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*. 95–107.
[22] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. 2017. KAML: A flexible, high-performance key-value SSD. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*. IEEE, 373–384.
[23] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. 2015. BlueDBM: An appliance for big data analytics. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. ACM, 1–13.
[24] Yangwook Kang, Yang-suk Kee, Ethan L. Miller, and Chanik Park. 2013. Enabling cost-effective data processing with smart SSD. In *Proceedings of the 29th Symposium on Mass Storage Systems and Technologies (MSST'13)*. IEEE, 1–12.
[25] Awais Khan, Hyogi Sim, Sudharshan S. Vazhkudai, and Youngjae Kim. 2021. MOSIQS: Persistent memory object storage with metadata indexing and querying for scientific computing. *IEEE Access* 9 (2021), 85217–85231.
[26] Jaeho Kim, Donghee Lee, and Sam H. Noh. 2015. Towards SLO complying ssds through OPS isolation. In *13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX, 183–189.
[27] Gunjae Koo, Kiran Kumar Matam, Te I, H. V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. 2017. Summarizer: Trading communication with computing near storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50'17)*. 219–231.
[28] Dongup Kwon, Dongryeong Kim, Junehyuk Boo, Wonsik Lee, and Jangwoo Kim. 2021. A fast and flexible hardware-based virtualization mechanism for computational storage devices. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC'21)*. 729–743.
[29] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[30] Willis Lang, Srinath Shankar, Jignesh M. Patel, and Ajay Kalhan. 2013. Towards multi-tenant performance SLOs. *IEEE Transactions on Knowledge and Data Engineering* 26, 6 (2013), 1447–1463.

[31] Chang-Gyu Lee, Hyeongu Kang, Donggyu Park, Sungyong Park, Youngjae Kim, Jungki Noh, Woosuk Chung, and Kyoung Park. 2019. iLSM-SSD: An intelligent LSM-Tree based key-value SSD for data analytics. In *Proceedings of the 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'19)*. IEEE, 384–395.

[32] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. 2019. Cognitive SSD: A deep learning engine for in-storage data retrieval. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC'19)*. USENIX, 395–410.

[33] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Wisckey: Separating keys from values in SSD-conscious storage. In *Proceedings of the File and Storage Technologies (FAST'16)*. USENIX, 133–148.

[34] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-tree database storage engine serving Facebook's social graph. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3217–3230.

[35] Donghyun Min and Youngjae Kim. 2021. Isolating namespace and performance in key-value SSDs for multi-tenant environments. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*. 8–13.

[36] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[37] Mohiuddin Abdul Qader, Shiwen Cheng, and Vagelis Hristidis. 2018. A comparative study of secondary indexing techniques in LSM-based NoSQL databases. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data*. 551–566.

[38] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. 2005. OpenDHT: A public DHT service and its SSEs. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. 73–84.

[39] Zhenyuan Ruan, Tong He, and Jason Con. 2019. Insider: Designing in-storage computing system for emerging high-performance drive. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC'19)*. USENIX, 379–394.

[40] Partho Sarthi, Kaushik Rajan, Akash Lal, Abhishek Modi, Prakhar Jain, Mo Liu, Ashit Gosalia, and Saurabh Kalikar. 2020. Generalized sub-query fusion for eliminating redundant I/O from big-data queries. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. USENIX, 209–224.

[41] David Shue, Michael J. Freedman, and Anees Shaikh. 2012. Performance isolation and fairness for multi-tenant cloud storage. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*. USENIX, 349–362.

[42] Swaminathan Sivasubramanian. 2012. Amazon dynamoDB: A seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 729–730.

[43] National Snow and Ice Data Center. 2016. World Glacier Inventory: Name, location, altitude, and area of every glacier on the planet. https://www.kaggle.com/nsidcorg/glacier-inventory.

[44] Satoru Watanabe, Kazuhisa Fujimoto, Yuji Saeki, Yoshifumi Fujikawa, and Hiroshi Yoshino. 2019. Column-oriented database acceleration using FPGAs . In *Proceedings of 2019 IEEE 35th International Conference on Data Engineering (ICDE'19)*. 686–697.

[45] ScaleFlux; Zhushi Cheng Alibaba; Ning Zheng ScaleFlux; Wei Li Wei Cao, Alibaba; Yang Liu, ScaleFlux; Peng Wang Wenjie Wu, Alibaba; Linqiang Ouyang, ScaleFlux; Zhenjun Liu Yijing Wang, Alibaba; Ray Kuan, and ScaleFlux Feng Zhu, Alibaba; Tong Zhang. 2014. POLARDB meets computational storage: Efficiently support analytical workloads in cloud-native relational database. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'14)*. USENIX, 29–41.

[46] Sung-Ming Wu, Kai-Hsiang Lin, and Li-Pin Chang. 2018. KVSSD: Close integration of LSM trees and flash translation layer for write-efficient KV store. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'18)*. IEEE, 563–568.

[47] Xavier. 2018. Tweets during Nintendo E3 2018 Conference. https://www.kaggle.com/xvivancos/tweets-during-nintendo-e3-2018-conference.

[48] Shuotao Xu, Thomas Bourgeat, Tianhao Huang, Hojun Koim, Sungjin Lee, and Arvind. 2020. AQUOMAN: An analytic-query offloading machine. In *Proceedings of the 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'20)*. 386–399.

[49] Yelp. 2021. A trove of reviews, businesses, users, tips, and check-in data! https://www.kaggle.com/yelp-dataset/yelp-dataset.

[50] Wei Zhang, Suren Byna, Chenxu Niu, and Yong Chen. 2019. Exploring metadata search essentials for scientific data management. In *Proceedings of the 26th International Conference on High Performance Computing, Data, and Analytics (HiPC'19)*. IEEE, 83–92.

[51] Wei Zhang, Suren Byna, Houjun Tang, Brody Williams, and Yong Chen. 2019. MIQS: Metadata indexing and querying service for self-describing file formats. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'19)*. Article 5, 24 pages.

[52] Xiao Zongshui, Lanju Kong, Qingzhong Li, and Pang Cheng. 2015. Global index oriented non-shard key for multi-tenant database. In *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*. IEEE, 831–836.

Received 29 June 2022; revised 15 February 2023; accepted 14 March 2023